Statistical Model Checking of LLVM Code

Louis-Marie Traonouez, Axel Legay, Dirk Nowotka, Danny Bøgsted Poulsen

Statistical Model Checking of LLVM Code.

Axel Legay[1], Dirk Nowotka[2], Danny Bøgsted Poulsen[2], and Louis-Marie Tranouez[1]

[1] Inria, Rennes
[2] Christian Albrechts Universität

Abstract. We present our work in providing Statistical Model Checking for programs in LLVM bitcode. As part of this work we develop a semantics for programs that separates the program itself from its environment. The program interact with the environment through function calls. The environment is furthermore allowed to perform actions that alter the state of the C-program - useful for mimicking an interrupt system. On top of this semantics we build a probabilistic semantics and present an algorithm for simulating traces under that semantics.. This paper also includes the development of the new tool component Lodin that provides a statistical model checking infrastructure for LLVM programs. The tool currently implement standard Monte Carlo algorithms and a simulator component to manually inspect the behaviour of programs. The simulator also proves useful in one of our other main contributions; namely producing the first tool capable of doing importance splitting on LLVM code. Importance splitting is implemented by integrating Lodin with the existing statistical model checking tool Plasma-Lab.

1 Introduction

The development of tools and techniques for verifying software systems behave correctly has been an active research area for more than two decades [6, 11, 13]. Common to the tools is that they take a model of the program along with a correctness criterion and explore every computational path in search of one violating it. This technique is commonly known as *Model Checking* [7, 8]. In case a model checking tool has found a violation, it provides a diagnostic trace for refining the model. Abstract models are useful to locate errors in the design of a program but less beneficial when an existing program is subjected to formal verification as that requires recreating the program in a formal model. A big problem in this regard is that old software may be left undocumented thereby making it difficult to create a representative model. In more recent years researchers developed Model Checking tools for real-life C-code [2, 4, 23]. The most prominent tools are possibly CPAchecker [4] and Divine [2]. CPAchecker analyses the C-source symbolically while Divine compiles C-source to LLVM [18] bitcode and analyses that explicitly. Zaks and Joshi [23] analysed LLVM bitcode by interfacing with the explicit-state model checker SPIN [13]. A limiting factor for Model Checking is the state-space explosion which drastically limits the size of

programs that can be handled by model checking. An alternative technique for verifying programs is *Runtime Verification* [3, 10], where the existing program is instrumented to expose its inner state at runtime to an observational unit. Runtime verification does not suffer from the state space explosion but does come with its own limitations: it can only draw conclusion on the traces it has seen, the instrumentation of the program alters the timing of events — a timing issues may depend on — and finally reinstantiating the program to a specific state is difficult making controlled experiments close to impossible.

Statistical Model Checking (SMC) [22] is a relative new technique that is a compromise between testing-based methods and model checking techniques. Like model checking SMC depends on a model of the program but does not perform an exhaustive exploration. Samples are instead generated from an underlying probability distribution and the probability of satisfying the property is estimated. Being simulation-based, SMC does not suffer from the state-space-explosion problem and being model-based SMC has complete control of its state and can thus reinstantiate to a given state. Statistical Model Checking thus proposes an alternative verification technique for C-code but no SMC tool with C as focus has been developed. In this work we develop the first Statistical Model Checking tool for C. Our tool, Lodin, operates on LLVM-code and thereby avoids the hassle of parsing and interpreting C and can focus on a simpler asssembly-like language. An added benefit of using LLVM as input is that the same C-code can be compiled with and without optimisations and our tool is applicable to both. A necessity for performing SMC is a probabilistic semantics for the program under verification and its environment. A first contribution of the current paper is the development of a probabilistic semantics of a LLVM program: at its core the semantics consist of the LLVM program given as a labelled transition system. The labels are function calls to an environment that implements functions outside the LLVM core language[3]. The environment is also responsible for assigning probabilities to individual transitions. Out probabilistic semantics is accompanied by the tool Lodin performing Statistical Model Checking under that semantics. Lodin also includes a simulator allowing the user to interactively inspect various thread interleavings.

A problem for simulation based techniques is that estimating rare properties result in generating a huge number of samples. Importance splitting [20] is an efficient technique for estimating rare properties. The general purpose SMC tool Plasma-Lab [5, 14, 19] implements importance splitting but lacks an LLVM simulator. In this paper we seamlessly integrate Lodin with Plasma-Lab and thus obtains the first tool capable of performing rare event simulation of LLVM programs. Besides providing means for estimating rare events, importance splitting is also a useful tool for recreating a behaviour pattern of a program. This is useful to asses the impact of changes to the program. We apply our tool to programs from the SV-Comp [1] benchmarks to asses the compatibility of Lodin with LLVM. We furthermore apply Lodin and the Plasma-Lab integration to 6 additional programs and see how importance splitting is effective when

[3] Could be various system calls

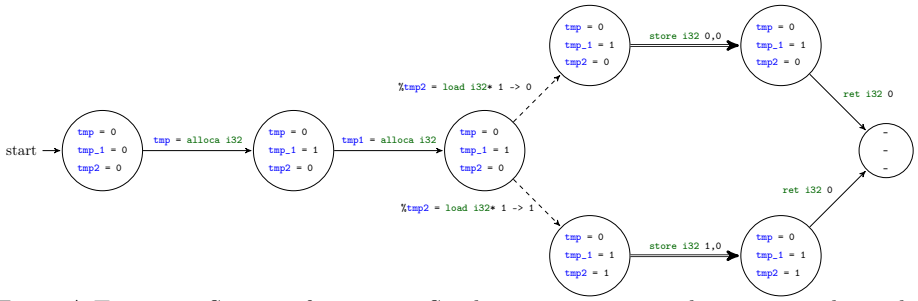

Fig. 1: A Transition System of a process. Single arrows are internal transitions that only affect the process itself. Dashed arrows correspond to input transitions, while double arrows correspond to output transitions.

classic Monte Carlo is not. In Section 2 we present our semantical framework along with our probabilistic semantics, followed by Section 3 which present the statistical model checking technique along with importance splitting for rare events. In Section 4 we introduce the new tool Lodin and the more mature tool Plasma-Lab and we discuss how these two tools are integrated. Finally, in Section 5 we present examples where we apply our developed tools.

2 Semantical Model

We consider a program given as a labelled transition system where labels indicate function calls to an environment. In this context, the environment controls the state of the memory and calls to the environment are operations such as `%reg = load i32 %loc` to load a 32 bit integer from memory location `%loc` into local register `%reg` and `store i32 %reg, i32* %loc` to store the value of `%reg` into memory location `%loc`. A `load` instruction is called an *input* as the local state depends on the outcome of the instruction.

Likewise, we consider a `store` instruction an *output* as it changes the state of the environment. Besides input and output instructions a program has internal operations that only changes its own local state (registers, control locations and so forth). As an example consider the LLVM program in Listing 1.1 and its associated transition system in Figure 1. The program allocates memory for two 32-bit integers on its local stack and store their memory addresses in local registers `%tmp` and `%tmp1`. Afterwards the value at location `%tmp1` is loaded into register `%tmp2`. Notice here that the program branches depending on the outcome of the load instruction. In practice there are branches for all $2^{32} - 1$ possible values of 32 bit numbers. For a set of names Σ, a function $\psi : \Sigma \to \mathbb{N}$ assigning arity to names and a finite domain D, we define the set of invocations as $\mathcal{F}^{\psi}(D, \Sigma) = \{a(d_1, \ldots d_n) \mid a \in \Sigma \wedge n = \psi(a) \wedge \forall i.d_i \in D\}$.

Listing 1.1: An example LLVM function.

```
1 ; Function Attrs: nounwind uwtable
2 define i32 @main() #0 {
3   % tmp = alloca i32, align 4
4   % tmp1 = alloca i32, align 4
5   % tmp2 = load i32, i32* %tmp1, align 4
6   store i32 %tmp2, i32* %tmp, align 4
7   ret i32 0
8 }
```

Definition 1. *A process is a tuple* $(D, \Sigma, \psi, S, s_0, \rightarrow)$ *where a)* D *is a finite domain, b)* Σ *is a finite set of names split into output* (Σ_o) *and input* (Σ_i) *such that* $\Sigma = \Sigma_o \cup \Sigma_i$ *and* $\Sigma_o \cap \Sigma_i = \emptyset$ *, c)* $\psi : \Sigma \rightarrow \mathbb{N}$ *assigns arity to names, d)* S *is a set of states, e)* $s_0 \in S$ *is the initial state, f)* $\rightarrow \subseteq S \times \big(\big(\mathcal{F}^{\psi}(D, \Sigma) \times (D \cup \{\lambda\})\big) \cup \{(\tau, \lambda)\}\big) \times S$*, where* $\lambda \notin D$ *and* $\tau \notin \mathcal{F}^{\psi}(D, \Sigma)$*, is a set of transitions with the restrictions that 1) if* $(s, (a, \lambda), s') \in \rightarrow$ *then* $a = \tau$ *or* $a \in \mathcal{F}^{\psi}(D, \Sigma_o)$ *and 2) if* $(s, (a, t), s') \in \rightarrow$ *and* $a \in \mathcal{F}^{\psi}(D, \Sigma_i)$ *then* $t \in D$*.*

As a short hand notation we write $s \xrightarrow{a \rightarrow t} s'$ whenever $(s, (a, t), s') \in \rightarrow$. The symbol τ indicates an internal transition and λ is a placeholder for "no value". The restrictions on the transition relation ensure λ is only used with τ- and output-transitions and that input transitions are always assigned of value from D. To make a process behave consistent with a real executing program we require that 1) the transition relation is successor-deterministic i.e. if $s \xrightarrow{a \rightarrow t} s'$ and $s \xrightarrow{a \rightarrow t} s''$ then $s' = s''$, 2) the transition relation is action-deterministic i.e. if $s \xrightarrow{a \rightarrow t} s'$ and $s \xrightarrow{a' \rightarrow t'} s''$ then $a = a'$ and 3) we expect that the transition relation is *fully non-deterministic* with respect to D meaning that if $s \xrightarrow{a \rightarrow t} s'$ and $t \in D$, then for all $j \in D$ there exists s'' a such that $s \xrightarrow{a \rightarrow j} s''$. An environment under which processes operate is a transition system awaiting synchronisations from processes and update the state according to outputs of processes and respond with values for the input synchronisations of a process. Formally, an environment is a tuple $\mathcal{E} = (D, \Sigma, \psi, S_e, s_0^e, \rightarrow_e)$ with components defined as for a process. Like for processes, we insist that an environment is successor-deterministic but additionally require that the environment is fully action-nondeterministic i.e. for any state s_e and any $a \in \mathcal{F}^{\psi}(D, \Sigma)$ there exists an $i \in D \cup \{\lambda\}$ such that $s \xrightarrow{(a->i)} s'$ for some s'. Also for all states $s_e \in S_e$ we insist there exists exactly one τ transition and that $s \xrightarrow{\tau \rightarrow \lambda} s$. Let $\mathcal{P} = (D, \Sigma, \psi, S, s_0, \rightarrow)$ be a process and let $\mathcal{E} = (D, \Sigma, \psi, S_e, s_0^e, \rightarrow_e)$ be an environment; then we call a tuple $\mathbf{s} = (\mathbf{s}, s_e)$ where $\mathbf{s} \in S^n$ for some $n \in \mathbb{N}$ and $s_e \in S_e$ for a network state. For a state vector $\mathbf{s} = (s_1, \ldots, s_n)$ we write $\mathbf{s}[i/s']$ for the updated state vector $(s_1, \ldots, s_{i-1}, s', s_{i+1}, \ldots, s_n)$. A network state $(\mathbf{s}, s_e)$ can transit to $(\mathbf{s}', s_e')$ by the i^{th} process doing an action $a \rightarrow t$ while synchronising with the environment - we write this transition as $(\mathbf{s}, s_e) \xrightarrow[i]{a \rightarrow t} (\mathbf{s}', s_e')$. Formally, the transition rule is defined as follows $(\mathbf{s}, s_e) \xrightarrow[i]{a \rightarrow t}_N (\mathbf{s}', s_e')$ if $\mathbf{s}[i] \xrightarrow{a \rightarrow t} s'$, $\mathbf{s}' = \mathbf{s}[i/s']$ and $s_e \xrightarrow{a \rightarrow t} s_e'$.

For synchronising programs, and guarding shared-memory from data races, programs usually rely on *locking* mechanisms. Processes attempting to lock an already locked lock is blocked until that lock is available for acquisition. In our semantics we consider that the environment keeps track of what processes are interrupted and which processes are active in its state space. In the following assume that there exists a function $\texttt{Active} : S_e \rightarrow 2^{\mathbb{N}}$ that extracts the indices of processes currently active. We can then define a new transition relation as $(\mathbf{s}, s_e) \xrightarrow[i]{a \rightarrow t} (\mathbf{s}, s_e)$ if $i \in \texttt{Active}(s_e)$ and $(\mathbf{s}, s_e) \xrightarrow[i]{a \rightarrow t}_N (\mathbf{s}, s_e)$.

Example 1. As an example of an environment, consider Figure 2. The environment has memory cells M_0 and M_1 mapped to values from a domain $D = \{0, 1\}$. A transition `load i32* 0 -> 1` is allowed if memory cell M_0 has the values 1. Obviously there would be a `load` instruction for both memory cells. Executing a `store i32 m,d` transition updates memory cell M_m to contain the value d. Here we also note that there is one `store` transition for all combinations of $m \in \{0, 1\}$ and $d \in D$. In Figure 2 we left out the τ-transitions. Finally, we note the environment state contains the id of active process - in this example there is only process active with id 0.

Environmental Transitions Computer systems often rely on an external program sending signals to a running program e.g. a signal informing some button has been pressed. For supporting this semantically we introduce an extra alphabet Σ_I and an extra transition relation $\rightsquigarrow \subseteq S \times \Sigma_I \times S$ for processes. We require that $\rightsquigarrow$ is successor-deterministic but fully non-deterministic with respect to Σ_I. For an environment, we introduce another transition relation $\rightsquigarrow^e \subseteq S_e \times \Sigma_I \times \mathbb{N} \times S_e$. The requirements to $\rightsquigarrow^e$ is that it must be successor-deterministic and if $(s_e, (k, i), {s_e}')$ then $i \in \texttt{Active}(s_e)$. This latter requirement ensures that an environment only sends signals to active processes. For a network state $(\mathbf{s}, s_e)$ we extend the transition relation with the rule $(\mathbf{s}, s_e) \xrightarrow[i]{k \to \lambda} (\mathbf{s}', {s_e}')$ if $k \in \Sigma_I$, $(s_e, k, i, {s_e}') \in \rightsquigarrow^e$, $(\mathbf{s}[i], k, s') \in \rightsquigarrow$ for some s' and $\mathbf{s}' = \mathbf{s}[i/s']$.

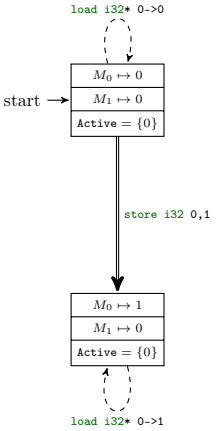

Fig. 2: Excerpt of the transition system for an

2.1 Probabilistic Semantics

In the following we define a probabilistic semantics refining the non-deterministic choices. If $\mathbf{s} = (\mathbf{s}, s_e)$ is a network state and $\mathcal{F}^\psi(D, \Sigma)$ is actions of the network, then for all $a \in \mathcal{F}^\psi(D, \Sigma) \cup \{\tau\}$ we let $\texttt{nd}(\mathbf{s})(a)$ extract the finite set of possible values the environment can respond with i.e. $t \in \texttt{nd}(\mathbf{s})(a)$ iff $s_e \xrightarrow{a \to t} s_e'$ for some s_e'. With this notation we now assume a set of probability mass functions: $\{\gamma^a_{\mathbf{s}} : \texttt{nd}(\mathbf{s})(a) \to \mathbb{R} \mid a \in \mathcal{F}^\psi(D, \Sigma) \cup \{\tau\}\}$ assigning probabilities to the various results. Likewise, we let $\texttt{intero}(\mathbf{s}) = \{(k, i) \in \Sigma_I \times \mathbb{N} | (s_e, k, i, s_e') \in \rightsquigarrow^e\}$ and assume a probability mass function $\gamma^{\mathtt{P}}_{\mathbf{s}} : \mathtt{N} \to \mathbb{R}$, where $\mathtt{N} = (\texttt{Active}(s_e) \times \{\_\}) \cup \texttt{intero}(s_e)$ assigning probabilities to the transitions of the network. Here _ is an unused symbol.

Let $\pi = \mathbf{s}_0, a_0, \mathbf{s}_1, a_1, \ldots, a_{n-1}\mathbf{s}_n$ be a finite sequence of states and actions with $\mathbf{s}_i = (\mathbf{s}_i, s_e^i)$ and $a \in \mathcal{F}^\psi(D, \Sigma) \cup \Sigma_I \cup \{\tau\}$, then the probability of observing this sequence from $\mathbf{s}$ is given by the expression

$$F_{\mathbf{s}}(\pi) = (\mathbf{s} \stackrel{?}{=} s_0) \cdot \sum_{(k,j) \in \mathcal{K}} \gamma^P_{\mathbf{s}_0}((k, j)) \cdot \begin{cases} \langle \mathbf{s}_0 \xrightarrow[k]{a_0} \rangle \cdot \left(\sum_{t \in \texttt{nd}(\mathbf{s}_0)(a_0)} \gamma^{a_0}_{\mathbf{s}_0}(t) \cdot F_{[\mathbf{s}]^{k,a_0,t}}(\pi^1) \right) & k \in \texttt{Active}(\mathbf{s}_0) \\ (k \stackrel{?}{=} a_0) \cdot F_{[\mathbf{s}]^{k,a_0,\lambda}}(\pi^1) & \text{otherwise} \end{cases}$$

where $\mathcal{K} = (\texttt{Active}(\mathtt{s}_o) \times \{\_\}) \cup \texttt{intero}(\mathtt{s}_0)$, $\mathtt{s} \stackrel{?}{=} \mathtt{s}'$ is 1 if $\mathtt{s} = \mathtt{s}'$ and 0 otherwise, $\langle \mathtt{s}_0 \xrightarrow[k]{a_0} \rangle$ is 1 if $\mathtt{s}_0 \xrightarrow[k]{a_0 \to t} \mathtt{s}'$ for some $\mathtt{s}'$ and t and 0 otherwise, $s \xrightarrow[k]{a \to t} [s]^{k,a_0,t}$ and $\pi^1 = s_1, a_1, \ldots, a_{n-1} s_n$.

A brief explanation of $F_{\mathtt{s}}$ may be in order: firstly it is checked whether $\mathtt{s}$ matches the first state of the sequence ($\mathtt{s}_0$). If that is the case, we sum over all possible actions,(k, j), the network can perform. If $k \in \texttt{Active}(\mathtt{s}_0)$ then we have selected to do a process transition and take into account the probability of the k$^{\text{th}}$ process doing action a_0, take the probability of the various output possibilities into account and finally multiply by the probability of seeing the remaining suffix of the sequence. In case $k \notin \texttt{Actives}_0$ we are considering an environmental transition and first check if $k = a_0$, then we take the probability of this transition into account and multiply by the probability of seeing the suffix.

3 Statistical Model Checking with Lodin

The verification technique we consider for verifying queries such as "What is the probability of reaching a state where ...?" is the approximate verification technique Statistical Model Checking (SMC) [21]. We split the discussion of SMC into two subsections: the first section presents the technique on a high-level and the second part presents the tool Lodin that provides SMC capabilities for LLVM programs.

3.1 Classical Monte Carlo Simulation

If F is a probability measure over the set of runs Ω and $\mathbf{1}_\phi : \Omega \to \{0, 1\}$ is an inidicator function for the property of interest, then the probability we are interested in is defined by the Lesbeque integral

$$\mathbb{P}(\phi) = \int_{\omega \in \Omega} \mathbf{1}_\phi(\omega) \, \mathrm{d}F,$$

which by classic probability theory can be estimated with the unbiased Monte-Carlo Estimator

$$\mathbb{P}(\phi) \approx \frac{1}{N} \sum_{i=0}^{N} \mathbf{1}_\phi(\omega_i),$$

Algorithm 1: Generating a run.

```
   Data: TimeBound: n
   Data: Initial state: s
   Result: A sequence ω
1  s_0 = s;
2  ω := s_0;
3  foreach i ∈ {0, ..., n − 1} do
4  |  (s, s_e) = s_i;
5  |  Let (j, k) ∼ γ^P_{s_i};
6  |  if k == _ then
7  |  |  Let a be such that
   |  |    s[i] --a→t/k--> s' for some t;
8  |  |  Let t ∼ γ^a_s;
9  |  |  Let s_i --a→t/j--> s_{i+1};
10 |  end
11 |  else
12 |  |  Let s_i --j→λ/k--> s_{i+1};
13 |  end
14 |  Let ω = ω : s_{i+1};
15 end
```

where each ω_i is distributed according to F, denoted $\omega_i \sim F$. From a practical perspective we are not only interested in an estimate but also a "measure" of how good that estimator is. If p is our estimator then we want to construct an interval $[p - \delta, p + \delta]$ for which we are α percent sure that the real probability

is contained within. We call α the confidence and $[p-\delta, p+\delta]$ a confidence-interval. We construct the confidence-interval using the method of Clopper and Pearson [9] for constructing confidence intervals for Binomial distributions. To visualise how one can use Clopper-Pearson interval for dynamically adjusting the number of runs needed, consider the graph in Figure 3. Here we for each generated run plot an 95%-confidence interval. Notice how the interval gets smaller as we produce more samples and as such we may continue generating samples until the confidence width is below a user specified threshold. Figure 3.

Obtaining Runs From the LLVM Program Samples are obtained from our probabilistic semantics with Algorithm 1: for each state $\mathtt{s}_i$ in the generated run we select a tuple (j,k) according to $\gamma^P_{\mathtt{s}_i}$ that represent the transition. In case k is _ then a process has been selected and we find the unique action a it will perform and select an output value from the environment according to γ^a. Finally, we create the new state $\mathtt{s}_{i+1}$ and append it to the run ω. If k is not _ then the selected transition is actually an environment transition and we just perform that transition to obtain the successor state and append the resulting state to the run.

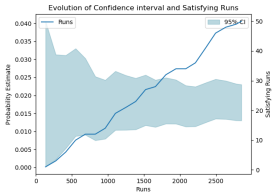

Fig. 3: Confidence interval over the number of runs.

3.2 Lodin

Lodin is a new tool for verifying programs given as LLVM-bitcode[4]. The tool implements explicit-state reachability checking algorithms and Monte-Carlo probability estimation techniques. Architecturally, Lodin consists of a *UI* front-end which a user interacts with. This front-end is responsible for loading a LLVM bit-code file, reading in a query from the user and setting up a suitable *System* and selecting an *Algorithm* for verifying the given query. As part of setting up the *System*, Lodin requires the user to provide it with a platform implementation. The platform provides an implementation inside Lodin of functions that are not given semantics by LLVM, e.g `pthread_create`. A Platform module is also responsible for creating *Environmental* transitions and assigning probabilities to different transitions - mimicking a scheduler mechanism. The System exposes an interface for algorithms to create an initial state, enumerating possible transitions from a state and for executing a given transition. When executing a transition the system makes a call to an interpreter for LLVM which may delegate function call to the *platform*.

Preparing files for use with Lodin Lodin requires programs to be compiled into LLVM bitcode. We can achieve this by compiling the program with `clang clang -emit-llvm -S -c file.c -o file.ll`. This is sufficient for programs with no external dependencies, but since properties are specified on the basis of the LLVM registers

[4] Available for download at `lodin.boegstedpoulsen.dk`

it may be necessary to run `opt instnamer file.ll -S -o fileN.ll` to generate the file `fileN.ll` in which the registers have been given names that LODIN can refer to. LODIN has extra requirements for programs with external dependencies: firstly a platform implementing external functions inside LODIN must exist, secondly the program must be compiled with header files distributed with the platform library. If the header files are located in `/path/to/includes` then programs should be compiled with the command

```
clang -nodefaultlibs -ffreestanding -fno-builtin -emit-llvm -S -c -I/path/
    to/includes file.c -o file.ll
```

ensuring clang compiles the program without using any of its built-in libraries and only rely on the header files included on the command line.

Using LODIN LODIN is a command line tool and is invoked by

```
./Lodin [options] file.ll query.q
```

where `[options]` includes options for selecting the platform, setting the random seed and so on. The `query.q` file contains a one line query. The possible SMC-based queries are generated by the below EBNF:

⟨*Query*⟩ ::= 'Pr' '[' '<=' ⟨*integer*⟩ ']' '(' '<>' ⟨*bool*⟩ ')'
| 'Estimate' '[' '<=' ⟨*integer*⟩ ',' ⟨*integer*⟩ ']' '{' 'max' ⟨*arith*⟩ '}'
| 'EnumStatesSMC' '<=' ⟨*integer*⟩ ⟨*integer*⟩

⟨*arith*⟩ ::= '@'⟨*processid*⟩ '.' ⟨*string*⟩ '.' ⟨*register*⟩ ';' ⟨*type*⟩ | ⟨*integer*⟩ ';'⟨*type*⟩
| ⟨*arith*⟩ ⟨*op*⟩ ⟨*arith*⟩

⟨*op*⟩ ::= '+' | '-' | '*' | '/'

⟨*type*⟩ :== 'ui8' | 'ui16' | 'ui32' | 'ui64'

⟨*bool*⟩ :== 'DataRace'
| '[' ⟨*processid*⟩'.'⟨*string*⟩ ']'
| ⟨*arith*⟩ ⟨*comp*⟩ ⟨*arith*⟩
| '('⟨*bool*⟩ '&&' ⟨*bool*⟩ '&&' ... '&&' ⟨*bool*⟩ ')'
| '(' ⟨*bool*⟩ '||' ⟨*bool*⟩ ... '||' ⟨*bool*⟩ ')'
| 'Exists' '('⟨*char*⟩ ')' '(⟨*bool*⟩')'
| 'Forall' '('⟨*char*⟩ ')' '(⟨*bool*⟩')'
|

⟨*comp*⟩ ::= '<' | '<=' | '==' | '>=' | '>' | '!='

⟨*processid*⟩ ::= ⟨*integer*⟩
| ⟨*char*⟩

⟨*register*⟩ ::= %⟨*string*⟩

An expression `@0.main.%tmp3;type` finds the register with name `%tmp3` in the function `main` of the zeroth process and evaluates it as a number with type `type`. An expression `[0.func]` looks at the zeroth process and checks whether it can call the function `func`. The Boolean expression `Exists (p)(<bool>)` checks whether there exists a process for which the Boolean expression is true. Any occurrence of `p` is replaced by an actual process during the evaluation. On the query side `Pr[<=500] (<> <bool>)` generates runs of length 500 and checks whether the Boolean expression is satisfied along those runs. Based on those runs it then estimates the probability of satisfying the Boolean expression along any random run. The exact number of runs is

automatically adjusted using the Clopper-Pearson interval. A query `Estimate [<=500,5000] {max <arith>}` generates 5000 runs each of 500 steps and estimates the expected maximal value of the expression during a run of the program. Finally, `EnumStatesSMC <=500 5000` generates 5000 runs each of 500 steps and counts the number of different states encountered during those simulations.

Example 2. Consider the program in Listing 1.2 where two threads attempt to calculate the 32^{nd} Fibonacci number. With LODIN we can estimate the expected number of i at the termination of the program. This is done using the query: `Estimate [<=5000,5000] {max @0.main.%tmp11}` . This query informs LODIN to generate 5000 runs, consisting of 5000 states each while observing the `%tmp11` register in the main function of the 0^{th} process and finally take the average of those 5000 observations. The result of this query is 438037. In addition to estimating the value, the query also outputs the values of each runs to a file which can be used to generate a histogram.

Listing 1.2: Calculating the Fibonacci Numbers.

```
1   extern void VERIFIERError()
        __attribute__ ((
        __noreturn__));
2
3   #include <pthread.h>
4
5   int i=1, j=1;
6
7   #define NUM 16
8   #define NULL 0
9
10
11  void *
12  t1(void* arg)
13  {
14    int k = 0;
15
16    for (k = 0; k < NUM; k++)
17      i+=j;
18
19    pthread_exit(NULL);
20  }
21
22  void *
23  t2(void* arg)
24  {
25    int k = 0;
26
27    for (k = 0; k < NUM; k++)
28      j+=i;
29
30    pthread_exit(NULL);
31  }
32
33  int
34  main(int argc, char **argv)
35  {
36    pthread_t id1, id2;
37
38  pthread_create(&id1, NULL, t1,
        NULL);
39    pthread_create(&id2, NULL, t2,
        NULL);
40    pthread_join (id1,NULL);
41    pthread_join (id2,NULL);
42
43    if (i == 2178309 || j ==
        2178309) {
44      ERROR: VERIFIERError();
45    }
46
47    return 0;
48  }
```

Example 3. The Fibonacci program considered in Example 2 is only correct if it at termination has found the 32^{nd} Fibonacci number (2178309). Using LODIN we can estimate the probability of having either $i = 2178309$ or $j = 2178309$ using `Pr[<=5000] (<> [0.VERIFIERError])` which asks for the probability that a state is reached where the 0^{th} process can call `VERIFIERError` - a call that is possible if either $i = 2178309$ or $j = 2178309$. Verifying the query with LODIN result in the probability being in the range $[0, 0.01]$ with confidence 0.95 and no satisfying executions found. As we will see later, the probability is 4.0e−6.

As can be seen from Example 2 rare events are problematic for SMC algorithms based around pure Monte Carlo simulation. For battling this problem we need to consider rare event simulation technique such as importance splitting.

4 Importance Splitting with Plasma-Lab

4.1 Importance Splitting

The unbiased Monte Carlo estimator is mostly useful when the property of interest is not "rare". For rare properties the number of runs needed to reliably estimate the probability increases rapidly — in fact it increases so rapidly that classical Monte-Carlo estimation becomes infeasible. To battle this problem a technique called importance splitting has been developed [20] and later extended to the statistical model checking approach [15]. In the following we consider properties that can be determined over a single run. For a property ϕ and run ω we write $\omega \models \phi$ if and only if ω satisfy ϕ. Consider we are interested in the property ϕ, and can find other properties $\phi_1, \phi_2, \ldots, \phi_k$ such that $\omega \models \phi \implies \omega \models \phi_k$ and for all i, $\omega \models \phi_i \implies \omega \models \phi_{i-1}$ for some $m \leq n$. If this is the case, then we can find the probability of ϕ as the product

$$\mathbb{P}(\phi) = \mathbb{P}(\phi_1) \cdot (\Pi_{i=2}^{n} \mathbb{P}(\phi_i | \phi_{i-1})) \cdot \mathbb{P}(\phi | \phi_n).$$

Estimating each of these conditional probabilities and multiplying them thus provides an estimate of the probability of ϕ. In practice a number of runs, N, is started from the initial state of the program. The fraction of runs that satisfy the first formula is then used as an estimate of the first conditional probability. The simulations that satisfied the first property is kept running, while the remaining ones are *restarted* from a satisfying one. In this way we continue until estimates for all the conditional probabilities has been found.

Example 4. Consider again the Fibonacci program in Listing 1.2 in which two threads attempt to calculate the 32nd Fibonacci number by the use of two global variables. If the program is functioning correct, either `i` or `j` has the value 2178309 by the end of the program. As noted earlier this is a rare property (probability is 4.0e−6) and classical Monte Carlo technique produces a probability of zero. For estimating the probability of $\phi = \Diamond(i = 2178309 \vee j = 2178309)$ one could use the intermediate formulas of the form $\phi_k = \Diamond(i + j == \mathcal{F}_{k+2})$, where $\mathcal{F}_1 = \mathcal{F}_2 = 1$ and for all other k, $\mathcal{F}_k = \mathcal{F}_{k-2} + \mathcal{F}_{k-1}$. Each ϕ_k is thus expressing that the program must have calculated the (k+2)th Fibonacci number correctly.

4.2 Plasma-Lab

Plasma-Lab [5] is a modular platform for statistical model-checking[5]. The tool offers a series of SMC algorithms, including advanced techniques for rare

[5] Available for download at `https://project.inria.fr/plasma-lab/`

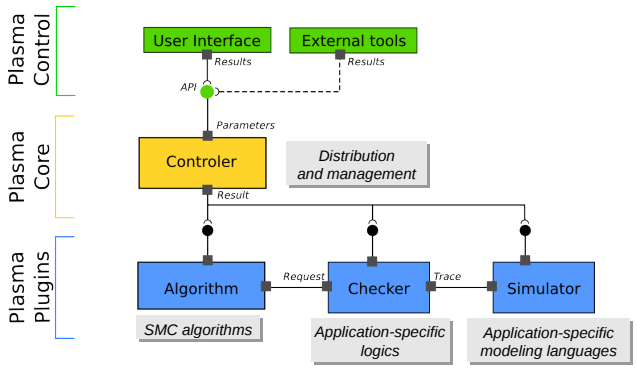

Fig. 4: Plasma-Lab architecture.

event simulation, distributed SMC, non-determinism, and optimization. They are used with several modeling formalisms and simulators. The main difference between Plasma-Lab and other SMC tools is that Plasma-Lab proposes an API abstraction of the concepts of stochastic model simulator, property checker (monitoring) and SMC algorithm. In other words, the tool has been designed to be capable of using external simulators, input languages, or SMC algorithms. This also allows us to create direct plug-in interfaces with external specification tools, without using extra compilers.

Plasma-Lab architecture is illustrated by the graph in Fig.4. The core of Plasma-Lab is a light-weight controller that manages the experiments and the distribution mechanism. It implements an API that allows to control the experiments either through user interfaces or through external tools. It loads three types of plugins: 1. algorithms, 2. checkers, and 3. simulators. These plugins communicate with each other and with the controller through the API.

In Plasma-Lab the decomposition of rare properties in a sequence of intermediate properties is generalised using a notion of score function over the model-property product automaton. Intuitively, a score function discriminates good paths from bad, assigning higher scores to paths that more nearly satisfy the overall property. The model-property product automaton is usually hidden in the implementation of the checker plugin. Therefore Plasma Lab includes a specific checker plugin for importance splitting that facilitates the construction of score functions. The plugin allows to write small observers automata to check properties over traces and compute the score function. These observers implement a subset of the Bounded Linear Temporal Logic presented in [17].

Plasma-Lab implements two rare event algorithms based on the importance splitting technique, a fixed level algorithm and an adaptive level algorithm [16]. The fixed level algorithm requires the user to define a monotonically increasing sequence of score values whose last value corresponds to satisfying the property. The adaptive algorithm finds optimal levels automatically and requires only the maximum score to be specified. Both algorithms estimate the probability of passing from one level to the next by the proportion of a constant number of

simulations that reach the upper level from the lower. New simulations to replace those that failed to reach the upper level are started from states chosen uniformly at random from the terminal states of successful simulations. The overall estimate is the product of the estimates of going from one level to the next.

For this paper, we have developed a simulator plugin for Plasma-Lab that interfaces with Lodin. This plugin is a pure wrapper around the simulator interface of Lodin. It communicates with the Lodin simulator via standard input and standard output. Lodin exposes the registers of all functions of the program to Plasma-Lab, and exposes Boolean variables corresponding to the `[0.func]` style propositions of Lodin. If the program has been compiled with debug symbols and without optimisations, Lodin also exposes the original C-source primitive type variables to Plasma-Lab. For supporting the importance splitting algorithm of Plasma-Lab, Lodin provides a *State-Tag* that Plasma-Lab uses to restart a simulation from that given state.

5 Examples

5.1 Support of C programs

Firstly we apply Lodin to a subset of the *pthread* examples of the Software Verification Competition (SV-Comp) benchmarks[6]. The point of this is mainly to assert if the engine adequately capture semantics of C-programs. The special `__VERIFIER_nondet_int` and `__VERIFIER_nondet_uint`[7] functions of the SV-Comp competitions are treated functions returning random integers. The original property from SV-Comp for all the programs was whether the function `__VERIFIER_error` could be called any timed during execution. The query we verify with Lodin is thus `Pr[<=5000] (<> Exists (p)([p.__VERIFIER_error]))`. The results are shown in Appendix A. In total, we applied Lodin to 44 programs from SV-Comp. Lodin exhibited an error for 3 of these program, while in 8 cases no satisfying runs were found while the property is true and in 3 cases Lodin found a satisfying runs while the property is false. The latter is unfortunate, but after manually inspecting the programs, it was discovered they relied on platform functions not implemented by Lodin (e.g. `sscanf` and `__VERIFIER_atomic_`). For the remaining programs Lodin found satisfying traces consistent with SV-Comp result.

5.2 Performance Test

In this section we apply our tools to 6 different C-programs. For all the programs we verify a reachability property with both Lodin and report the running times as well as the reported probability estimates. For properties that are rare we also apply Plasma-Lab. The are shown in Table 1 and Table 2.

[6] Obtained from `https://github.com/sosy-lab/sv-benchmarks`
[7] Functions to return non-deterministic integers

Fibonacci We have parameterised the program Listing 1.2 such that instead of being fixed to calculate the 32^{nd} we can scale it to find the n^{th} fibonacci number. The property of interest is still whether ther $zero^{th}$ process can call `VERIFIERError`. In the results we refer to programs of this kind by `fib_n` where `n` is what Fibonacci number is being calculated.

Gossip This program is inspired by gossiping protocols where processes exchange messages to distribute knowledge in the network. In this particular example, the processes communicate by writing to `commWars` which the processes copy to their own `secret` variable. Once a process has all messages that process terminates. The main thread waits for all processes to terminate and checks if all processes knows all secrets and if not calls `VERIFIERError`. At the the end of execution the main threads calls `VERIFIERFinished`. Unforunately, there is an error in the program and it may not terminate - thus we are interested in the probability that `VERIFIERFinished` is called by the main thread i.e. `Pr[<=5000] (<> [0.VERIFIERFinished])`. We have parameterised the program in terms of number of processes and in the results refer to these program by `gossip_n` where `n` is the number of processes.

Petersons Algorithm Petersons algorithm is a classic mutual exclusion protocol that does not rely on any locking mechanisms. Two processes, `petersons1` and `petersons2`, are both attempting to reach their critical section - abstracted by a call to a function `crit`. The two processes are communicating through a shared array of `flags` and a special `turn` variable. The property we are interested in is whether the two processes can execute the `crit` function simultaneously. We both consider a correct implementation and a faulty implementation. The property we check in this program is `Pr[<=5000] (<> ([1.crit] && [2.crit]))`.

Robot Control This is a larger example in which a robot is placed onto a 9×9 two-dimensional grid and has to locate the top-right tile. The robot will break if it tries to leave the grid. The robot divides its control into two threads: one thread is continuously monitoring whether the goal has been reached and queries whether it can move up, down, left or right and sets global variables for the outcome of these. The second thread selects a direction that is safe to move and moves in that given direction. The direction is selected randomly according. In case the robot accidentally leaves the grid the program calls `VERIFIERCrash` and enters an infinite loop. If the search for the goal is successful the program calls `VERIFIERDone`. The query we check for in this program is `Pr[<=5000] (<> [0.VERIFIERDone])`

PTrace This example is a simplified version of the PTrace privilege escalation attack that could be performed on earlier versions of the Linux Kernel [12]. In our simplified version two processes are started of which one (`syscaller`) performs a system call, captured by the call to `syscall`. Another process (`ptracer`) *attaches* to the `syscaller` process via `ptrace` and awaits a call to `syscall`. When the syscall occurs, the `ptracer` process immidiately calls `poketext` - which we use as an abstract way of saying that the process injects code to the syscaller process. Signal handlers are used by the kernel to inform the `ptracer` that interesting events has happened.

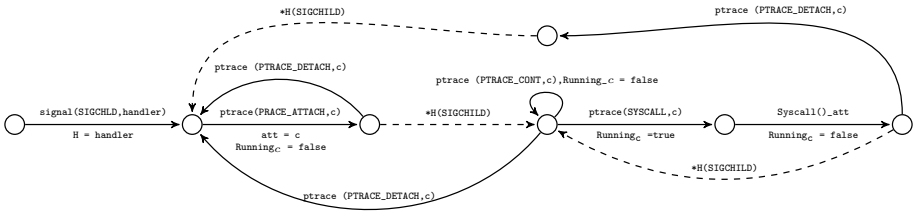

Fig. 5: Environment for PTrace Example. Solid lines correspond to calls from the running program while dashed lines are environmental transitions.

This works by the `ptracer` registering a handler function at the kernel, and the kernel switching the thread to execute that function when the signal is raised. For this example we implemented a platform module for Lodin implementing the `sycall`,`syscallend`,`signal`,`ptrace` functions. In Figure 5 we give the state graph controlling the interaction with the `ptracer` process : first it awaits the `ptracer` to register a signalling function for SIGCHLD signals, afterwards it awaits for `ptracer` to attach to another process with id c. During the attaching the process c is stopped and an environmental transition (SIGCHLD) is executed (which forces the `ptracer` process to execute the function pointed to by *H). If `ptracer` call `ptrace(PTRACE_CONT,c)` then the process c is restarted - c is also restarted if `ptrace(PTRACE_SYSCALL,c)` is called but then the environment will subsequently stop c when it performs a system call and inform the `ptracer` process about the system call. The query we check with Lodin is `Pr[<=5000] (<> [1.poketext])` which designates that we reach the point where we could inject code. If the attack is unsuccessful then the `ptracer` thread loops infinitely and would not reach the `poketext` function call.

Lock-Free-Stack This is an implementation of a lock-free stack of integers which two threads are popping and pushing integers onto. Initially the main thread pushes 10 onto the stack then the two threads are started of which one thread first pops an element off the stack and then pushes 5. The other thread only pushes 20. Provided the stack is correctly implemented, the possible stack contents when the program terminates are $(20, 5)$,$(5, 10)$ or $(5, 20)$. Unfortunately there is a problem with the implementation that allows a result $(5, 100)$. We can verify the probability of this happening by checking if `@0.main.\%tmp17` is ever 100[8]. In Lodin we verify the query `\lstinline{Pr[<=5000] (<> [0.VERIFIERDone])`. As can be seen from the results in Table 1 Lodin estimates the probability to $[0.00, 0.01]$ and has found no satisfying runs. As it turns out this is just because the property is rare, as we will see by by finding traces with Plasma-Lab. The problem with the program is that two variables are updated in the wrong order while popping an element from the stack which results in the two threads changing the same variable - a shared variable that should never be accessed simultaneously by two

[8] `@0.main.%tmp17` is a register used in the `main` function to print out the stack-content at program termination

Program	Runs	Satisfying	CI	Time (s)
fib/fib_4.ll	19 242	2789	[0.14 , 0.15]	3.80
fib/fib_8.ll	7453	370	[0.04 , 0.05]	2.26
fib/fib_16.ll	299	0	[0.00 , 0.01]	0.16
fib/fib_32.ll	299	0	[0.00 , 0.01]	0.28
ptrace/ptrace.ll	33 249	10 412	[0.31 , 0.32]	22.82
gossip/gossip_2.ll	34 470	11 575	[0.33 , 0.34]	219.68
gossip/gossip_3.ll	13 187	1229	[0.09 , 0.10]	94.41
gossip/gossip_4.ll	8450	481	[0.05 , 0.06]	66.30
petersons/petersonsBug.ll	10 870	816	[0.07 , 0.08]	1.64
petersons/petersons.ll	299	0	[0.00 , 0.01]	0.05
robot/robot.ll	2507	38	[0.01 , 0.02]	109.65
stack/stack.ll	299	0	[0.00 , 0.01]	7.16

Table 1: Lodin results. The Runs columns is total number of generated runs, Satisfying is the number of satisfying runs while the CI column is a 95% confidence interval.

threads. The observer in Plasma-Lab works by first getting thread 2 to change the first variable in the `pop` function and restricting thread 1 from executing its change to the variable. Once accomplished, we swap the roles and let thread 1 continue until it has changed the value of the shared variable while holding back thread 2. To inform Plasma-Lab where in their execution the threads were we had to inject extra statements into the program.

6 Conclusion

In this paper we have presented the first statistical model checking framework for LLVM-program. As part of this we have presented a probabilistic semantics for LLVM programs and a simulator for LLVM programs using that semantics. Another contribution of the paper is the new tool Lodin providing statistical model checking features for LLVM programs and an integration of this new tool component with Plasma-Lab — providing importance splitting to LLVM code as well. In the paper we validated that Lodin can verify the concurrency programs — modulo the implemented POSIX calls and SV-Comp special functions. Furthermore, Lodin and the Plasma-Lab were run on six additional programs. The integration between Plasma-Lab and Lodin showed its worth as the estimation of rare properties were made possible. In the future we intend to further develop the range of programs our toolsets can be applied to by implementing a POSIX compliant interface for Lodin. Furthermore, we intend to ease the burden of defining platform modules - possibly by creating a domain-specific language for defining these. On the more theoretical side we intend to use our tools for reasoning on the impact of data races with respect to software correctness.

Program	Levels	Probability	Time (s)
fib/fib_16.ll	7	1.5e−3	18.20
fib/fib_32.ll	14	4.0e−6	51.66
stack/stack.ll	13	3.86e−15	530.58

Table 2: Plasma-Lab Importance Splitting Results. The algorithm was run with a budget of 1000 runs per level.

References

[1] Competition on software verification (sv-comp). https://sv-comp.sosy-lab.org/2018/. Accessed: 2017-10-19.

[2] J. Barnat, L. Brim, and P. Rockai. Towards LTL model checking of unmodified thread-based C & C++ programs. In A. Goodloe and S. Person, editors, *NFM*, volume 7226 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2012. ISBN 978-3-642-28890-6. doi:10.1007/978-3-642-28891-3_25.

[3] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, Sept. 2011. ISSN 1049-331X. doi:10.1145/2000799.2000800.

[4] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. ISBN 978-3-642-22109-5. doi:10.1007/978-3-642-22110-1_16. URL `https://doi.org/10.1007/978-3-642-22110-1_16`.

[5] B. Boyer, K. Corre, A. Legay, and S. Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In K. R. Joshi, M. Siegle, M. Stoelinga, and P. R. D'Argenio, editors, *QEST*, volume 8054 of *Lecture Notes in Computer Science*, pages 160–164. Springer, 2013. doi:10.1007/978-3-642-40196-1_12.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002. ISBN 3-540-43997-8. doi:10.1007/3-540-45657-0_29.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, pages 52–71, 1981.

[9] C. J. Clopper and E. S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934.

[10] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems. In *SEFM*, volume 7041 of *LNCS*, pages 204–220, 2011.

[11] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2014. ISBN 978-3-642-54861-1. doi:10.1007/978-3-642-54862-8_13.

[12] J. Goubault-Larrecq and J. Olivain. A smell of orchids. In M. Leucker, editor, *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers*, volume 5289 of *Lecture Notes*

in Computer Science, pages 1–20. Springer, 2008. ISBN 978-3-540-89246-5. doi:10.1007/978-3-540-89247-2_1. URL `https://doi.org/10.1007/978-3-540-89247-2_1`.

[13] G. J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[14] C. Jegourel, A. Legay, and S. Sedwards. A Platform for High Performance Statistical Model Checking – PLASMA. In C. Flanagan and B. König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 498–503. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28755-8. doi:10.1007/978-3-642-28756-5_37.

[15] C. Jegourel, A. Legay, and S. Sedwards. Importance splitting for statistical model checking rare properties. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *LNCS*, pages 576–591. Springer, 2013.

[16] C. Jegourel, A. Legay, and S. Sedwards. An effective heuristic for adaptive importance splitting in statistical model checking. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 143–159. Springer, 2014.

[17] C. Jegourel, A. Legay, S. Sedwards, and L.-M. Traonouez. Distributed verification of rare properties using importance splitting observers. *ECEASST*, 72, 2015.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] A. Legay, S. Sedwards, and L. Traonouez. Plasma lab: A modular statistical model checking platform. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 9952 of *Lecture Notes in Computer Science*, pages 77–93, 2016. doi:10.1007/978-3-319-47166-2_6.

[20] G. Rubino and B. Tuffin. *Rare Event Simulation using Monte Carlo Methods*. Wiley, 2009.

[21] H. L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon University, 2005.

[22] H. L. S. Younes, M. Z. Kwiatkowska, G. Norman, and D. Parker. Numerical vs. statistical probabilistic model checking. *STTT*, 8(3):216–228, 2006.

[23] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *SPIN*, volume 5156 of *Lecture Notes in Computer Science*, pages 325–342. Springer, 2008. ISBN 978-3-540-85113-4. doi:10.1007/978-3-540-85114-1_22.

A Full Results for the SV_Comp models

In the below table we show all the models from *SV_Comp* that Lodin has been applied to. ☹-rows indicate models for which Lodin found traces satisfying the property (while *SV_Comp* says the property is not satisfied), 😐-rows are programs where no traces were found by Lodin while the property is satisfied. In the latter case this is possible because the property is unlikely (given our semantics) while the prior is more severe. However, we have discovered that the ☹-programs relied on platform functions not implemented in Lodin (One example is the use of `__VERIFIER_atomic_` functions that signal to the verifier this function should be performed atomically). ♿-rows are models where Lodin exhibits an error during the verification.

	Model	Runs	Sat	CI	Time(s)	SV_Comp
☺	pthread/lazy01_-false-unreach-call.ll	31221	8763	[0.28, 0.29]	16.13	👎
😐	pthread/reorder_-5_false-unreach-call.ll	299	0	[0.00, 0.01]	0.02	👎
☺	pthread/singleton_-with-uninit-problems_true-unreach-call.ll	299	0	[0.00, 0.01]	0.06	👍
☺	pthread/queue_-longest_false-unreach-call.ll	17959	2395	[0.13, 0.14]	144.28	👎
☺	pthread/sigma_-false-unreach-call.ll	35345	12518	[0.35, 0.36]	47.21	👎
☺	pthread/stack_-longest_true-unreach-call.ll	299	0	[0.00, 0.01]	3.36	👍
😐	pthread/stack_-longest_false-unreach-call.ll	299	0	[0.00, 0.01]	3.53	👎
☺	pthread/stateful01_-true-unreach-call.ll	299	0	[0.00, 0.01]	0.07	👍
☺	pthread/queue_-ok_true-unreach-call.ll	299	0	[0.00, 0.01]	0.78	👍
😐	pthread/stack_-longer_false-unreach-call.ll	299	0	[0.00, 0.01]	3.44	👎

☺ pthread/fib_-bench_true-unreach-call.ll	299	0	[0.00, 0.01]	0.04	👍
☺ pthread/indexer_-true-unreach-call.ll	299	0	[0.00, 0.01]	4.99	👍
☺ pthread/stack_-true-unreach-call.ll	299	0	[0.00, 0.01]	0.32	👍
😐 pthread/stack_-false-unreach-call.ll	299	0	[0.00, 0.01]	0.33	👎
☺ pthread/queue_-false-unreach-call.ll	17721	2326	[0.13, 0.14]	67.41	👎

😐 pthread/fib_-bench_false-unreach-call.ll	299	0	[0.00, 0.01]	0.04	👎
☺ pthread/fib_-bench_longest_-true-unreach-call.ll	299	0	[0.00, 0.01]	0.03	👍
☺ pthread/queue_-longer_false-unreach-call.ll	18372	2518	[0.13, 0.14]	149.18	👎
😐 pthread/fib_-bench_longer_-false-unreach-call.ll	299	0	[0.00, 0.01]	0.04	👎
☺ pthread/sync01_-true-unreach-call.ll	299	0	[0.00, 0.01]	1.50	👍

☺ pthread/bigshot_-s2_true-unreach-call.ll	299	0	[0.00, 0.01]	0.05	👍
☺ pthread/bigshot_-s_true-unreach-call.ll	299	0	[0.00, 0.01]	0.04	👍
😐 pthread/fib_-bench_longest_-false-unreach-call.ll	299	0	[0.00, 0.01]	0.03	👎
☺ pthread/singleton_2-false-unreach-call.ll	21372	3527	[0.16, 0.17]	4.12	👎

☺ pthread/fib_bench_longer_true-unreach-call.ll	299	0	[0.00, 0.01]	0.04	👍
☺ pthread/stack_longer_true-unreach-call.ll	299	0	[0.00, 0.01]	3.43	👍
☺ pthread/bigshot_p_false-unreach-call.ll	38612	19340	[0.50, 0.51]	4.71	👎
♿ pthread/reorder_2_false-unreach-call.ll	0	0	[0.00, 0.00]	0.01	👎
☺ pthread/stateful01_false-unreach-call.ll	299	299	[0.99, 1.00]	0.07	👎
😐 pthread/elimination_backoff_stack_false-unreach-call.ll	299	0	[0.00, 0.01]	0.06	👎
☺ pthread/queue_ok_longer_true-unreach-call.ll	299	0	[0.00, 0.01]	2.15	👍
♿ pthread/twostage_3_false-unreach-call.ll	0	0	[0.00, 0.00]	0.01	👎
☺ pthread/queue_ok_longest_true-unreach-call.ll	299	0	[0.00, 0.01]	2.15	👍
☺ pthread-atomic/qrcu_false-unreach-call.ll	10724	793	[0.07, 0.08]	91.06	👎
☺ pthread-atomic/time_var_mutex_true-unreach-call.ll	299	0	[0.00, 0.01]	0.06	👍
☺ pthread-atomic/lamport_true-unreach-call.ll	299	0	[0.00, 0.01]	0.09	👍
♿ pthread-atomic/scull_true-unreach-call.ll	0	0	[0.00, 0.00]	0.00	👍

☹	pthread-atomic/qrcu_-true-unreach-call.ll	10989	835	[0.07, 0.08]	96.29	👍
☺	pthread-atomic/read_-write_lock_false-unreach-call.ll	38263	17308	[0.45, 0.46]	13.41	👎
☺	pthread-atomic/peterson_-true-unreach-call.ll	299	0	[0.00, 0.01]	0.06	👍
☹	pthread-atomic/gcd_true-unreach-call_-true-termination.ll	26809	20835	[0.77, 0.78]	255.20	👍
☺	pthread-atomic/dekker_-true-unreach-call.ll	299	0	[0.00, 0.01]	0.21	👍
☹	pthread-atomic/read_-write_lock_true-unreach-call.ll	38245	17252	[0.45, 0.46]	13.52	👍
☺	pthread-atomic/szymanski_-true-unreach-call.ll	299	0	[0.00, 0.01]	2.81	👍

B Program Texts

B.1 Petersons

Besides the correct implementation in Listing 1.3 we also consider a buggy-version where we have changed `*(opt->mflag)= 1;` in line 15 to `*(opt->mflag)= 0;`

Listing 1.3: petersons.c

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void crit ();
5
6  typedef struct {
7    int *mflag;
8    int *oflag;
9    int* turn;
10 }Options;
11
12 void* petersons1 (void* opti) {
13   Options* opt = (Options*)opti;
14
15   *(opt->mflag) = 1;
16   *(opt->turn) = 1;
17
18         while (*(opt->oflag)  &&
                *(opt-> turn) == 1)
19         {
20           // busy wait
21         }
22   // critical section
```

```
  crit ();
  // end of critical section

  *(opt->mflag) = 0;

  return 0;
}

void* petersons2 (void* opti) {
   Options* opt = (Options*)opti;

   *(opt->mflag) = 1;
   *(opt->turn) = 0;

   while (*(opt->oflag)  && *(opt
        -> turn) == 0)
         {
           // busy wait
         }
   // critical section
   crit ();
   // end of critical section

   *(opt->mflag) = 0;

   return 0;
}

int main () {
  int flags[2];
  int turn;

  Options opt1;
  opt1.mflag = &flags[0];
  opt1.oflag = &flags[1];
  opt1.turn = &turn;

  Options opt2;
  opt2.mflag = &flags[1];
  opt2.oflag = &flags[0];
  opt2.turn = &turn;

  pthread_t t1;
  pthread_t t2;

  pthread_create (&t1,0,petersons1
       ,&opt1);
  pthread_create (&t2,0,petersons2
       ,&opt2);

  pthread_join (t1,0);
  pthread_join (t2,0);


  return 0;
}
```

B.2 Gossip

Listing 1.4: gossip.c

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

#ifndef PROCS
#define PROCS 2
#define ALLKNOWN 3
#endif

int  VERIFIERError (){}
void VERIFIERFinished () {}

typedef struct {
  int* secret;
  int* commVars;
  int id;
}setup;

void* protocol (void* inp) {
  setup* set = (setup*) inp;
  while (*set->secret != ALLKNOWN)
        {
         int selectedRecv = rand ()
              % PROCS;
         set->commVars[selectedRecv
              ] = *set->secret;
         if (set->commVars[set->id]
              != 0) {
           *set->secret |= set->
                commVars[set->id];
           set->commVars[set->id] =
                0;
         }
  }
  return 0;
}

void main () {
  int secretsKnown[PROCS];
  int commVar[PROCS];
  pthread_t threads[PROCS];
  setup sets[PROCS];
  for (int i = 0; i < PROCS;i++) {
        secretsKnown[i] = 1 << i;
        commVar[i] = 0;
        sets[i].secret = &
             secretsKnown[i];
        sets[i].commVars = commVar
             ;
        sets[i].id = i;
        printf ("%i",secretsKnown[
             i]);
  }


  for (int i= 0; i< PROCS;i++) {
        pthread_create (&threads[i
             ],0,protocol,&sets[i
             ]);
  }

  for (int i = 0;i < PROCS; i++) {
        pthread_join (threads[i
             ],0);
  }
```

```
54 
55   for (int i = 0; i< PROCS; i++) {
56         if (secretsKnown[i] !=
               ALLKNOWN)
57           VERIFIERError ();
58   }
59 
60   VERIFIERFinished ();
61 }
```

B.3 PTrace

Listing 1.5: ptrace.c

```
1  #include <stdint.h>
2  #include <signal.h>
3
4  void poketext ();
5
6  int syscaller () {
7    syscall ();
8    syscallend ();
9  }
10
11 int32_t c = 0;
12
13 void handler (int i) {
14   c++;
15 }
16
17 int ptracer () {
18   signal (SIGCHLD,handler);
19   ptrace (PTRACE_ATTACH,0);
20   while (c < 1);
21   ptrace (PTRACE_SYSCALL,0);
22   while (c < 2);
23   poketext ();
24   ptrace (PTRACE_CONT,0);
25   ptrace (PTRACE_DETACH,0);
26 }
```

B.4 Robot

Listing 1.6: robot.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <pthread.h>
4  #include <time.h>
5  #include "interface.h"
6
7  void done () {}
8
9  typedef struct {
10   int up,down,left,right,goal,sense;
11 }Comm;
12
13 void* observer (void* inp){
14   Comm* comm = (Comm*) inp;
15   while (comm->sense) {
16         comm->goal = goal ();
17         comm->up = lookUp ();
18         comm->down = lookDown ();
19         comm->right = lookRight ();
20         comm->left = lookLeft ();
21   }
22   return 0;
23 }
24
```

```
void* runner (void* inp) {
  Comm* comm = (Comm*) inp;
  time_t t;
  srand((unsigned) time(&t));

  int i = 0;
  while (!comm->goal ) {
        i++;
        int32_t dir = rand () %4;

        switch (dir) {
        case 0:
          if (comm->up)
                moveUp ();
          break;
        case 1:
          if (comm->down)
                moveDown ();
          break;
        case 2:
          if (comm->left)
                moveLeft ();
          break;
        case 3:
          if (comm->right)
                moveRight();
          break;
        }
  }
  comm->sense = 0;
  return 0;
}

void gridReady () {}

int main () {
  Comm comm;
  initGrid ();
  gridReady ();
  comm.sense = 1;
  comm.goal = 0;
  pthread_t senser;
  pthread_t runnerr;
  pthread_create (&senser,0,observer,&comm);
  pthread_create (&runnerr,0,runner,&comm);

  pthread_join (senser,0);
  pthread_join (runnerr,0);
  VERIFIERDone ();
  return 0;
}
```

Listing 1.7: platform.c

```
#include <pthread.h>
#include "interface.h"

#define WIDTH 9
#define HEIGHT 9

pthread_mutex_t mutex;

int32_t grid[WIDTH*HEIGHT];

int32_t xPos = 0;
int32_t yPos = HEIGHT -1;

int32_t VERIFIERDone () {while (1) {}}
```

```
15 int32_t VERIFIERCrash () {while (1) {} return 0;}
16
17 void lock () {
18   pthread_mutex_lock (&mutex);
19 }
20
21 void unlock () {
22   pthread_mutex_unlock (&mutex);
23 }
24
25 void initGrid () {
26   for (int32_t w = 0; w< WIDTH; w++) {
27         for (int32_t h = 0; h < HEIGHT;h++) {
28           grid[WIDTH*h+w] = 2;
29         }
30   }
31
32   grid[WIDTH*0+WIDTH-1] = 1;
33   pthread_mutex_init(&mutex,0);
34 }
35
36 int32_t innerLookUp () {
37   if (yPos - 1 >= 0) {
38         return grid[WIDTH*(yPos-1)+xPos];
39   }
40   return 0;
41 }
42
43 int32_t innerLookDown () {
44   if (yPos + 1 < HEIGHT) {
45         return grid[WIDTH*(yPos+1)+xPos];
46   }
47   return 0;
48 }
49
50
51 int32_t innerLookRight () {
52    if (xPos + 1 < WIDTH) {
53         return grid[WIDTH*(yPos)+xPos+1];
54   }
55   return 0;
56 }
57
58 int32_t innerLookLeft () {
59   if (xPos - 1 >= 0) {
60         return grid[WIDTH*(yPos)+xPos-1];
61   }
62   return 0;
63 }
64
65 int32_t moveUp () {
66   lock ();
67   if (innerLookUp ()) {
68         yPos --;
69   }
70   else
71         VERIFIERCrash ();
72   unlock ();
73 }
74
75 int32_t moveDown ()
76   {
77         lock ();
78         if (innerLookDown ()) {
79           yPos ++;
80         }
81         else
82           VERIFIERCrash ();
```

```
83          unlock ();
84    }
85  int32_t moveLeft () {
86    lock ();
87    if (innerLookLeft ()) {
88          xPos --;
89    }
90    else
91          VERIFIERCrash ();
92    unlock ();
93  }
94  int32_t moveRight () {
95    lock ();
96    if (innerLookRight ()) {
97          xPos++;
98    }
99    else
100         VERIFIERCrash ();
101   unlock ();
102 }
103
104
105
106 int32_t lookUp () {
107   lock ();
108   int32_t val = innerLookUp ();
109   unlock ();
110   return val;
111 }
112
113
114
115 int32_t lookDown () {
116   lock ();
117   int32_t val =  innerLookDown ();
118   unlock ();
119   return val;
120 }
121
122
123
124 int32_t lookLeft () {
125   lock ();
126   int32_t val = innerLookLeft ();
127   unlock ();
128 }
129
130
131 int32_t lookRight ()   {
132   lock ();
133   int32_t val = innerLookRight ();
134   unlock ();
135   return val;
136 }
137
138 int32_t goal () {
139   return grid[yPos*WIDTH+xPos] == 1;
140 }
141
142 int32_t getX () {return xPos;}
143 int32_t getY () {return yPos;}
```

Listing 1.8: interface.h

```
1  #include <stdint.h>
2
3  void initGrid ();
4
```

```
int32_t moveUp ();
int32_t moveDown ();
int32_t moveLeft ();
int32_t moveRight ();

int32_t lookUp ();
int32_t lookDown ();
int32_t lookLeft ();
int32_t lookRight ();
int32_t goal ();
int32_t getX ();
int32_t getY ();
int32_t VERIFIERDone ();
int32_t VERIFIERCrash ();
```

B.5 Stack

Listing 1.9: stack.c

```
#include <stdio.h>
#include <pthread.h>

struct Element {
  int i;
  struct Element* next;
  int index;
};

struct Element* head;

const int capacity = 20;

struct Element elemsBuf[capacity];
int free[capacity];

void init () {
  head = 0;
  for (int i = 0; i < capacity; i
       ++) {
        elemsBuf[i].i = 100;
        elemsBuf[i].next = 0;
        elemsBuf[i].index = i;
        free[i] = 1;
  }

}

void gotFree () ;
void inLoop () ;
void inIf () ;
void gotFree1 () ;
void gotFree2 () ;
void gotFree3 () ;

struct Element* getFree () {
  while (1) {
        int selected = -1;
        gotFree ();
        for (int i = 0; i <
             capacity; i++) {
          inLoop ();
          if (free[i]) {
                gotFree1 ();
                selected = i;
                break;
          }
        }
        gotFree2 ();
        if (selected >= 0) {
          inIf ();
          if (
               __sync_bool_compare_and_swap
                (&free[selected
               ],1,0))
                gotFree3 ();
                return &elemsBuf[
                     selected];
        }
  }
}

void pushed () ;
void pushed1 () ;
void pushed2 ();

void push (int i) {
  struct Element* elem = getFree
       ();
  struct Element* hh;
  pushed ();
  elem->i = i;
  pushed2 ();
  do {
        hh = head;
        pushed1 ();
        elem->next = hh;

  }while (!
       __sync_bool_compare_and_swap
        (&head,hh,elem));
  pushed3 ();
  printf ("Pushed %i-%i\n",elem->i
       ,elem->index);
}

void  popped () ;
void  poppedHead () ;
void  popped1 () ;
void  popped2 () ;


int pop () {
  struct Element* elem = 0;
  do {
        elem = head;
        poppedHead ();
        if (!elem)
```

```
89            return -1;
90    }while (!
          __sync_bool_compare_and_swap
           (&head,elem,elem->next));
91    popped ();
92    int i = elem->i;
93    free[elem->index] = 1;
94    popped1 ();
95    popped2 ();
96    elem->i = 100;
97
98    return i;
99  }
100
101
102
103 void* thread1 (void* n) {
104   push (20);
105 }
106
107 void* thread2 (void* n) {
108   pop ();
109   push (5);
110 }
111
112 int main () {
113   init ();
114   pthread_t  t1,t2;
115   push (10);
116   pthread_create (&t1,0,thread1,0)
          ;
117   pthread_create (&t2,0,thread2,0)
          ;
118   pthread_join (t1,0);
119   pthread_join (t2,0);
120   struct Element* h = head;
121   while (h) {
122         printf ("%i(%i) ",h->i,h->
              index);
123         h = h->next;
124   }
125
126   printf ("\n");
127 }
```